Artificial Malware Immunization based on Dynamically Assigned Sense of Self

Xinyuan Wang[1] and Xuxian Jiang[2]

[1] Department of Computer Science
George Mason University, Fairfax, VA 22030, USA
email: xwangc@gmu.edu
[2] Department of Computer Science
North Carolina State University University, Raleigh, NC 27606, USA
email: jiang@cs.ncsu.edu.edu

Abstract. Computer malwares (e.g., botnets, rootkits, spware) are one of the most serious threats to all computers and networks. Most malwares conduct their malicious actions via hijacking the control flow of the infected system or program. Therefore, it is critically important to protect our mission critical systems from malicious control flows.

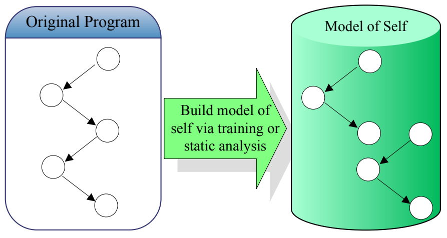
Inspired by the self-nonself discrimination in natural immune system, this research explores a new direction in building the artificial malware immune systems. Most existing models of self of the protected program or system are passive reflection of the existing being (e.g., system call sequence) of the protected program or system. Instead of passively reflecting the existing being of the protected program, we actively assign a unique mark to the protected program or system. Such a dynamically assigned unique mark forms *dynamically assigned sense of self* of the protected program or system that enables us to effectively and efficiently distinguish the unmarked nonself (e.g., malware actions) from marked self with no false positive. Since our artificial malware immunization technique does not require any specific knowledge of the malwares, it can be effective against new and previously unknown malwares.

We have implemented a proof-of-concept prototype of our artificial malware immunization based on such dynamically assigned sense of self in Linux, and our automatic malware immunization tool has successfully immunized real-world, unpatched, vulnerable applications (e.g., Snort 2.6.1 with over 140,000 lines C code) against otherwise working exploits. In addition, our artificial malware immunization is effective against return-to-libc attacks and recently discovered return-oriented exploits. The overall run time performance overhead of our artificial malware immunization prototype is no more than 4%.

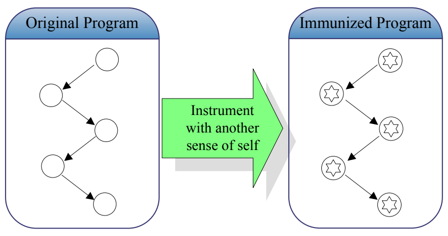
Keywords Malware Immunization, Control Flow Integrity, Sense of Self.

1 Introduction

Despite recent advances in malware defense, computer malware (e.g., virus, worm, botnets, rootkits, trojans, spyware, keyloggers) continues to pose serious threats to the trustworthiness of all computers and networks.



(a) Traditional sense of self



(b) Dynamically assigned sense of self

Fig. 1. Self-nonself discrimination based on traditional sense of self vs the proposed dynamically assigned sense of self

In addition to being more damaging, modern malware is becoming increasingly stealthy and evasive to avoid detection. For example, Agobot, a major bot family with thousands of variants in-the-wild, contains self protection code to periodically detect and remove popular anti-malware processes and sniffers (e.g., tcpdump, ethereal) on the infected hosts. Return-oriented programming [31] enables malware to use the trusted, benign code (e.g., libc) to conduct all kinds of malware activities. Such a technique not only enables malware activities without any code injection or calling any libc function, but also raises the fundamental question on whether the trusted, benign code is free of malware computation. All these have made people wonder whether we could possibly win the battle on malware and what we can expect from malware detection.

On the other hand, our natural immune systems are very effective in protecting our body from the intrusion by (almost endless) variations of pathogens. Our immunities depend on the ability to distinguish our own cells (i.e., "self") from all others (i.e., "nonself") [9]. Such a self-nonself discrimination is so fundamental to our immunities that immunology is widely regarded as "the science of self-nonself discrimination" [23].

Forrest et al. [15, 14] have pioneered in applying the idea of immunology to computer security. If we consider the uninfected computer system as "self" and malwares as "non-self", protecting uninfected computer systems from invading malwares is very similar to protecting our body from invading pathogens from the perspective of self-nonself discrimination. Specifically, if we can effectively and efficiently distinguish the actions of the uninfected computer system (self) from the actions of malwares (nonself), we can build effective and efficient artificial malware immunization to "immunize" the uninfected computer system from the malicious actions of malwares.

In this paper, we explore a new direction in building the artificial malware immunization capabilities based on a new way to model and monitor the normal behavior (i.e. self) of the protected program or system. As shown in Figure 1(a), most existing models of self of the protected program or system [15, 32, 29, 13, 18, 8] are passive reflection of the existing being (e.g., system call sequence) of the protected program or system. Instead of passively reflecting the existing being of the protected program, we actively assign a unique mark to the protected program or system via program instrumentation as shown in Figure 1(b). Such a program instrumentation "immunizes" the protected program or system, and the dynamically assigned unique mark forms the *dynamically* *assigned sense of self* of the instrumented program or system, which enables us to effectively and efficiently distinguish the unmarked nonself (e.g., malware actions) from marked self with no false positive.

Based on the dynamically assigned sense of self, we have built a framework of artificial malware immunization that has the following salient features:

- **Transparent:** Our artificial malware immunization is able to immunize user space applications transparently without changing any source code of the applications to be immunized.
- **Effective:** Our artificial malware immunization is effective against various control flow hijacking attacks (e.g., buffer overflow, return-to-libc, return-oriented exploit [31]). Since our artificial malware immunization does not require any specific knowledge of malwares, it can be effective against previous unknown malwares. In addition, it is effective against malwares obfuscated by mimicry attack [33].
- **Virtually no false positive:** In theory, our artificial malware immunization framework will never falsely accuse any normal action of the immunized program or system to be malware action. In practice, our implementation of the artificial malware immunization achieves virutally no false positive in detecting the nonself malware action.
- **Efficient:** Our artificial malware immunization incurs neglectable run-time overhead over the original program or system.
- **New real-time malware forensics support:** Our artificial malware immunization is able to locate and identify the first and all shell commands and system calls issued by the malware in real-time. This unique feature enables new malware forensics capabilities (e.g., accurately locating the offending input by malware) that were not possible in the past.

We have implemented a prototype of our artificial malware immunization in Linux. Our automatic malware immunization tool has successfully instrumented the GNU standard C library (glibc 2.5 of about 1 million lines of C code) and immunized real-world, unpatched, vulnerable applications (e.g., Snort 2.6.1 of over 140,000 lines of C code) against otherwise working exploits. In addition, our artificial malware immunization is effective against return-to-libc attacks and recently identified return-oriented exploits [31]. Without special optimization effort, our automatic malware immunization prototype is able to immunize vulnerable applications with no more than 4% overall run-time overhead.

The rest of the paper is organized as follows. In section 2, we present the design and implementation of the artificial malware immunization. In section 3, we empirically evaluate the effectiveness and efficiency of our artificial malware immunization. In section 4, we review related works. We conclude in section 5.

2 Design and Implementation

2.1 Goals and Assumptions

Inspired by the self-nonself discrimination in natural immune systems, we build an artificial malware immunization framework based on dynamically assigned sense of self. Instead of trying to detect statically if any particular object (e.g., program, file, packet) contains any malware, the primary goal of our artificial malware immunization is to prevent malwares from doing harm to the protected program or system at run-time. Specifically, our artificial malware immunization aims to automatically immunize otherwise vulnerable applications and systems (as shown in Figure 1(b)) such that (1) it is capable of defending and detecting both control flow attacks and data flow attacks [11]; (2) it has no false positive in detecting the malware; (3) it is more efficient in run-time checking than current generation of models of self; (4) it enables new malware forensics capabilities.

Here we assume that the program or system to be immunized is free of malware. We build the artificial malware immunization upon the trust on the operating system kernel, and we assume that there is no malware beneath the operating system kernel. In other words, there is no lower layer malware (e.g., hardware based rootkit, VMBR).

In principle, we can immunize both the control flow and its data access via dynamically assigned sense of self. Since most existing malware infection involves control flow hijacking [3], we focus on immunizing the control flow of the vulnerable programs in this paper. Since most real-world vulnerable applications are written in C, we focus on how to immunize systems written in C and leave the immunization of programs written in other (e.g., script) languages as a future work.

Ideally, we want to be able to mark each instruction of the program with dynamically assigned sense of self so that we can detect the first instruction executed by the malware (nonself). However, checking each instruction would incur prohibitively high overhead. On the other hand, a compromised application could hardly do any harm without using system calls (e.g., write) [15, 24]. Therefore, we can prevent all application level control flow hijaking malwares from doing any harm if we can detect and block their first system call. For this reason, we choose to define the dynamically assigned sense of self of control flow at the granularity of system call.

Unlike previous models of self where the sense of self is based on system call sequence, our proposed dynamically assigned sense of self of control flow is essentially a unique mark assigned to each system call. The unique mark assigned to each system call provides another sense of self to the system call that is orthogonal to the system call sequence based sense of self.

2.2 Overall Architecture of Artificial Malware Immunization

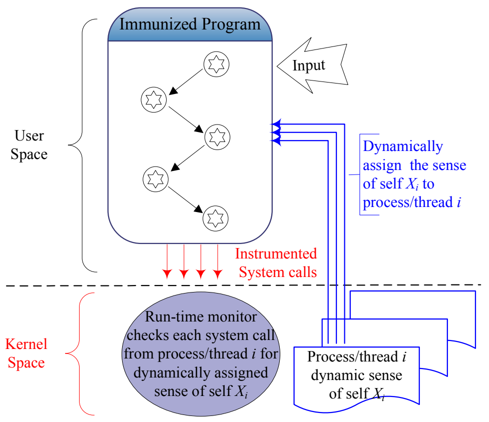
Our artificial malware immunization framework consists of two functional components: 1) the malware immunization tool that can transparently immunize user space programs (e.g., applications, libraries) offline; and 2) the malware immunization infrastructure that supports run-time malware immunization and forensics.

Figure 2 illustrates the overall architecture of the artificial malware immunization based on dynamically assigned sense of self. To immunize an otherwise vulnerable program based on dynamically assigned sense of self, we first use program instrumentation tools (e.g., extended gcc) to statically instrument the vulnerable program into the immunized program such that each system call invocation in the program contains the unique

[3] We do recognize that there are certain attacks [11] that do not hijack control flow but rather change critical data flow

mark that will be dynamically assigned by the artificial malware immunization infrastructure (i.e. the instrumented OS kernel) at run-time. When the immunized program is loaded to run, the instrumented OS kernel first creates a process/thread for the program, and then randomly generates and stores a unique mark X_i as the dynamically assigned sense of self of the control flow of the process/thread. The instrumented OS kernel puts X_i into the run-time environment (e.g., envp[]) for the process/thread before transferring the control to the newly created process/thread. Therefore, each process/thread of any immunized program will have its own unique dynamically assigned sense of self X_i.

When the process/thread invokes any system call, it marks the system call with the unique X_i passed from the instrumented OS kernel. When the instrumented OS kernel receives the system call, it first looks up the unique mark X_i for the process/thread that has invoked the system call and then checks if the received system call has the correct unique mark X_i. Only the system call that has the correct unique mark X_i (i.e., dynamically assigned sense of self of control flow) is considered self. Since each system call invoked from the immunized program is instrumented to have the correct unique mark X_i, the instrumented OS kernel will recognize the system calls invoked by the immunized program as self. On the other hand, if some malware has somehow (e.g., via buffer overflow) gained control and started executing its malicious code or the chosen libc function (e.g., `system()`), the system calls invoked by the malware do not have the correct mark X_i since the walware is not part of the original immunized code. Therefore, the instrumented OS kernel is able to catch the first and all the system calls invoked by the malware unless the malware has somehow found out and used the unique mark X_i in its system calls. We will discuss how to make it difficult for the malware to recover the dynamically assigned sense of self X_i in section **??**.



Immunized Program
Input
User Space
Dynamically assign the sense of self X_i to process/thread i
Instrumented System calls
Kernel Space
Run-time monitor checks each system call from process/thread i for dynamically assigned sense of self X_i
Process/thread i dynamic sense of self X_i

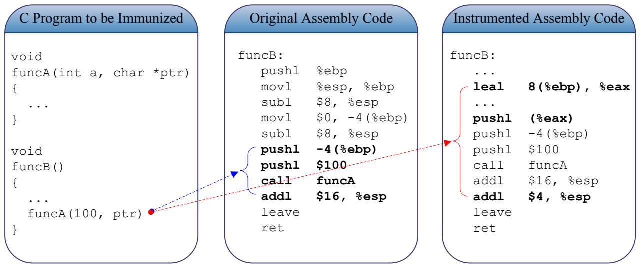
Fig. 2. Artificial malware immunization based on dynamically assigned sense of self of control flow

2.3 Transparently Immunizing Programs

Given a potentially vulnerable program, we want to be able to transparently immunize it so that it will be immune from control flow hijacking attacks while keeping the original semantics unchanged. We achieve this goal of transparent malware immunization via static program instrumentation.

Assuming we have access to the source code of the program to be immunized, we have implemented the transparent malware immunization by extending gcc 3.4.3 such that it will generate object code or executable with built-in support of the dynamically assigned sense of self. Note our artificial malware immunization does not require any changes on the source code of the program to be immunized.

Specifically, we use an unsigned number X_i randomly generated by the malware immunization infrastructure (i.e. operating system kernel) to represent the dynamically assigned sense of self of a process or thread i. The immunized program is instrumented to transparently pass the dynamically assigned number X_i as the rightmost parameter, in addition to all existing parameters, to all function and system call invocations. Essentially, this is equivalent to adding an implicit rightmost parameter to all functions and system calls. Because the default C functional calling convention passes parameters from right to left, which supports variable number of parameters, the extra implicit rightmost parameter (i.e., the dynamically assigned sense of self X_i) to C function calls will not change the original semantics of the C function.



```
void
funcA(int a, char *ptr)
{
  ...
}

void
funcB()
{
  ...
  funcA(100, ptr)
}
```

```
funcB:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $8, %esp
   movl    $0, -4(%ebp)
   subl    $8, %esp
   pushl   -4(%ebp)
   pushl   $100
   call    funcA
   addl    $16, %esp
   leave
   ret
```

```
funcB:
   ...
   leal    8(%ebp), %eax
   ...
   pushl   (%eax)
   pushl   -4(%ebp)
   pushl   $100
   call    funcA
   addl    $16, %esp
   addl    $4, %esp
   leave
   ret
```

Fig. 3. Transparently Immunizing Programs via Instrumented Implicit Rightmost Parameter

Figure 3 illustrates the transparent malware immunization of programs by showing the assembly code of the original and the instrumented function invocations.

Instrumenting C Functions with Variable Number of Parameters

```
int __printf (const char *format, ...)
{
  va_list arg;
  int done;

  va_start (arg, format);
  done = vfprintf (stdout, format, arg);
  va_end (arg);

  return done;
}
```

Certain C functions have variable number of parameters. For example, the above snippet from file `stdio-common/printf.c` of glibc 2.5 shows that function `__printf` has variable number of parameters in addition to the fixed parameter `format`. Since our transparent malware immunization is achieved through static program instrumentation offline, the dynamic nature of variable number of parameters imposes new challenges that need special handling when immunizing the program with variable number of parameters.

By parsing the invocation of `__printf`, we can find the exact number of actual parameters the caller of `__printf` passed to each instance of `__printf` call. Therefore, we can pass the dynamically assigned sense of self X_i as the rightmost parameter, in addition to variable number of actual parameters, when calling function `__printf`.

The problem arises when we are inside `__printf`, trying to pass the dynamically assigned sense of self X_i further down to function `vfprintf`. Specifically, we need to accurately locate the position of X_i passed by the caller of `__printf` from inside `__printf` before we can pass it further down to functions called inside `__printf`. Since `__printf` may have variable number of actual parameters passed and the exact number of actual parameters may be determined by run-time input from user, there is no way for us to statically find out the exact number of actual parameters passed by the caller from inside `__printf`.

To address this issue, we instrument the function (e.g., `__printf`) of variable number of parameters with code that searches for the dynamically assigned sense of self X_i passed by the function caller. Specifically, we introduce a random canary word used as the marker of the right location of the dynamically assigned sense of self. The following code snippet shows how we can search for the right location of the dynamically assigned sense of self X_i from inside the functions of variable number of parameters with the help of the random canary word.

```
     movl %ebp, %eax
     addl x, %eax
     movl $0, %ebx
.L1: cmpl canary, 4(%eax)
     je .L2
     cmpl n, %ebx
     jge .L2
     incl %ebx
     addl $4, %eax
     jmp .L1
.L2: push canary
     push (%eax)
```

Note only functions of variable number of parameters need this special handling. The canary word can be randomly chosen by the compiler, and different programs can be instrumented with different canary word. Therefore, the adversary is not able to identify the canary word used in one instrumented binary unless he has access to that instance of the instrumented binary.

When some parameter passed by the caller of function of variable number of parameters happens to be the same as the random canary word chosen by the compiler, the instrumented function or system call may use the wrong number as the dynamically assigned sense of self. In this case, legitimate system call may be mistakenly regarded as nonself. Note this autoimunity problem is due to the limitation of our current implementation rather than the model of dynamically assigned sense of self. In practice, we can effectively make the probability of such autoimmunity extremely small by using large size canary. With n-bit long canary word, the probability of autoimmunity for the current immunization implementation of functions of variable number of parameters is no more than $\frac{1}{2^n}$.

2.4 Artificial Malware Immunization Infrastructure

Besides transparently immunizing programs, artificial malware immunization requires run-time support from the infrastructure. The malware immunization infrastructure is responsible for 1) generating the random, dynamically assigned sense of self for each newly created process/thread; and 2) checking each system call from interested process/thread for the dynamically assigned sense of self and acting according to policy specified for each process/thread at run-time.

To build the malware immunization infrastructure, we have instrumented Linux kernel 2.6.18.2. Specifically, we have added the following two integer fields to the `task_struct`

```
unsigned int DASoS;
unsigned int DASoSflag;
```

When the instrumented kernel creates a new process/thread, it will generate a random number and store it in field `DASoS` as the dynamically assigned sense of self of the newly created process/thread. Before the instrumented kernel transfers the control to the newly created process/thread, it puts the dynamically assigned sense of self in the run-time environment at user space so that the newly created process/thread can use it when invoking any function or system call.

Before the immunized user space program learns its dynamically assigned sense of self from its run-time environment at the beginning of its execution, the first few function and system calls of the newly started process/thread will not have the correct dynamically assigned sense of self. Once the newly started process/thread learns its dynamically assigned sense of self from its run-time environment, all the subsequent function and system calls will have the correct dynamically assigned sense of self.

By default, field `DASoSflag` is unset meaning that the process/thread will not be checked by the instrumented kernel for the dynamically assigned sense of self. A user level utility allows user to inform the malware immunization infrastructure about which process/thread it should check for the dynamically assigned sense of self and what it should do once any nonself system call is detected.

Note the user level utility only changes the global state of the malware immunization infrastructure, and it does not change the per process/thread field `DASoSflag`. Such a design is to accommodate the first few system calls that do not have the correct dynamically assigned sense of self. When the instrumented kernel is set to check a particular process/thread, it will not do any specified action (e.g., report, block) on the process/thread unless the per process/thread field `DASoSflag` is set. The instrumented kernel will set the per process/thread field `DASoSflag` once it sees the first system call that has the correct dynamically assigned sense of self. It will further mark the process/thread as nonself by setting field `DASoSflag` once any nonself system call is detected.

One nice feature of this two level organization of the flag of the dynamically assigned sense of self is that it allows inheritance of the flag. Specifically, the child process will inherit field `DASoSflag` from the parent process. Such a inheritance of flag enables us to effectively track those nonself actions and new processes spawned by the malware process. For example, assume process A has been compromised and controlled by the malware, and we want to do live forensics on process A and see what actions the malware is taking. Once the malware issues a command (e.g., *ls*) which spawns a new process B, the newly created process B will have field `DASoSflag` set as nonself since it is inherited from process A, which is already marked as nonself by the malware immunization infrastructure. Therefore, our artificial malware immunization infrastructure is able to identify and collect all the nonself actions of the identified malware in real-time.

To enable the passing of the dynamically assigned sense of self from user space to kernel via system calls, we have instrumented the system call entrance interface in glibc 2.5. This turns out to be the most time consuming task in building the artificial malware immunization infrastructure. We have manually instrumented 12 assembly files and several header files, most of which dealing with low-level interface (e.g., getcontext, swapcontext) with the kernel.

3 Evaluation

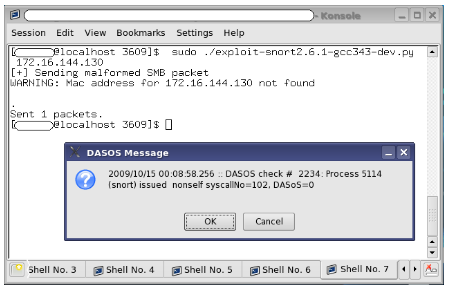
To evaluate the effectiveness and efficiency of our artificial malware immunization framework based on dynamically assigned sense of self, we have implemented the malware immunization infrastructure by instrumenting the Fedora Core 6 Linux kernel and glibc 2.5. To support transparent immunization of programs, we instrumented gcc 3.4.3 such that it will "immunize" the generated the executable or library with support of dynamically assigned sense of self. In this section, we show how our artificial malware immunization framework is able to detect, block malware actions in real-time, and what real-time malware forensics is can support.

3.1 Effectiveness of the Artificial Malware Immunization

To demonstrate the practical value of our artifical malware immunization, it is desirable to use real-world exploits on real-world vulnerble applications. However, many known real-world exploits are based the same attack vector (e.g., buffer overflow), and there could be new attack vectors (e.g., return-oriented programming [31]) that have not (yet) been used by any malware in the wild. Therefore, simply experimenting with more known real-world malwares does not necessarily tell more about the effectiveness of our artificial malware immunization.

Since our artifical malware immunization detects and blocks the malware action based on dynamically assigned sense of self on each system call invoked by the immunized program, the way in which the malware exploits the immunized program – be it stack overflow, heap overflow, integer overflow, format string overflow, or some other unkown attack vector – is not important. Here we assume the malware can somehow seize the control flow of the vulnerable program and start running its malicious logic (e.g., via code injection, return-to-libc or return-oriented programming [31]). The effectiveness of our artificial malware immunization is measured by how well it can detect and block the system calls invoked by the infecting malware and how well it can immunize those otherwise vulnerable applications.

To demonstrate the effectiveness of our malware immunization framework, we have chosen to experiment with two real world exploits on real-world applications: ghttpd buffer overflow exploit [1], Snort DCE/RPC packet reassembly buffer overflow exploit [3]. While ghttpd is a very compact Web server with less than 800 lines of C code, Snort [2] is the most popular open source network based IDS with over 140,000 lines of C code. Specifically, Snort is claimed to be the "most widely deployed IDS/IPS technology worldwide" with over 250,000 registered users. Immunizing a real world



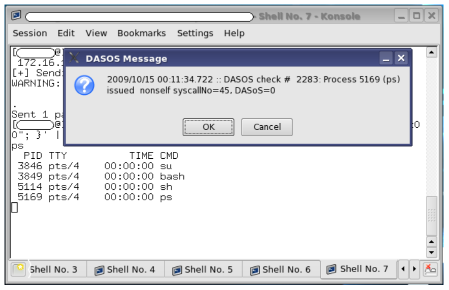
```
[[redacted]@localhost 3609]$  sudo ./exploit-snort2.6.1-gcc343-dev.py
 172.16.144.130
[+] Sending malformed SMB packet
WARNING: Mac address for 172.16.144.130 not found

.
Sent 1 packets.
[[redacted]@localhost 3609]$ 
```

DASOS Message

2009/10/15 00:08:58.256 :: DASOS check # 2234: Process 5114 (snort) issued nonself syscallNo=102, DASoS=0

(a) Detect the first nonself system call when Snort 2.6.1 was compromised



[redacted] - Shell No. 7 - Konsole
Session Edit View Bookmarks Settings Help

```
[[redacted]@
 172.16.
[+] Send
WARNING:

.
Sent 1 p
[[redacted]@                                                            :0
0"; }' |
ps
  PID TTY          TIME CMD
 3846 pts/4    00:00:00 su
 3849 pts/4    00:00:00 bash
 5114 pts/4    00:00:00 sh
 5169 pts/4    00:00:00 ps
```

DASOS Message

2009/10/15 00:11:34.722 :: DASOS check # 2283: Process 5169 (ps) issued nonself syscallNo=45, DASoS=0

OK
Cancel
Shell No. 3
Shell No. 4
Shell No. 5
Shell No. 6
Shell No. 7

(b) Detect the first nonself system call of the "ps" command issued from the remote shell of the compromised Snort 2.6.1

Fig. 4. Real-time detection of nonself system call and nonself command of the Snort 2.6.1 exploit

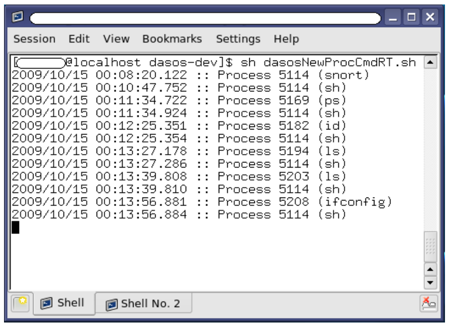
application of such a size and popularity would be a goot test on how practical our artificial malware immunization is.

In addition to traditional code injection exploits, we have experimented with recently discovered return-oriented exploit [31]. To cover those potential but not (yet) identified exploits, we have used the buffer overflow benchmark developed by Wilander and Kamkar [36] which contains 20 synthetic attacks based on vaious stack overflow and heap overflow. Our artificial malware immunization is able to block all the 20 synthetic attacks. In the rest of this section, we focus on describing the experiments on the two real-world exploits and the recently discovered return-oriented exploit [31].

In our experiments, we have first "immunized" the otherwise vulnerable applications with our transparent malware immunization tool (i.e., instrumented gcc 3.4.3) and then have launched the working exploits against the "immunized" applications. Our artificial malware immunization framework is able to detect, block all the tested exploits in real-time. In addition, it supports preliminary but real-time forensics analysis on the live malware actions.

Real-time Detection of Malware Infection We set the malware immunization infrastructure to be in detection mode when we have launched the working exploit on the immunized Snort 2.6.1. Figure 4 shows the real-time detection of nonself system call and nonself command of the working exploit on Snort 2.6.1. Specifically, Figure 4(a) shows the detection of the first nonself system call # 102 invoked by the compromised Snort process 5114. Figure 4(b) shows the detection of the first nonself system call invoked by the command "ps" issued from the remote root shell gained from the compromised Snort 2.6.1. As shown in the screenshot, the original Snort process 5114 has been changed to a "sh" process by the exploit. Although the "ps" command issued by the attack created a new process 5169, our artificial malware immunization infrastructure was able to correctly identify the newly spawned process 5169 as nonself and report it in real-time.

As shown in the "DASOS Message" message boxes in Figure 4, all the real-time detection messages are timestamped with precision up to millisecond.



```
[[redacted]@localhost dasos-dev]$ sh dasosNewProcCmdRT.sh
2009/10/15 00:08:20.122 :: Process 5114 (snort)
2009/10/15 00:10:47.752 :: Process 5114 (sh)
2009/10/15 00:11:34.722 :: Process 5169 (ps)
2009/10/15 00:11:34.924 :: Process 5114 (sh)
2009/10/15 00:12:25.351 :: Process 5182 (id)
2009/10/15 00:12:25.354 :: Process 5114 (sh)
2009/10/15 00:13:27.178 :: Process 5194 (ls)
2009/10/15 00:13:27.286 :: Process 5114 (sh)
2009/10/15 00:13:39.808 :: Process 5203 (ls)
2009/10/15 00:13:39.810 :: Process 5114 (sh)
2009/10/15 00:13:56.881 :: Process 5208 (ifconfig)
2009/10/15 00:13:56.884 :: Process 5114 (sh)
```

Fig. 5. Real-time forensics of the commands issued and the processes spawned by the attacker after compromising Snort 2.6.1

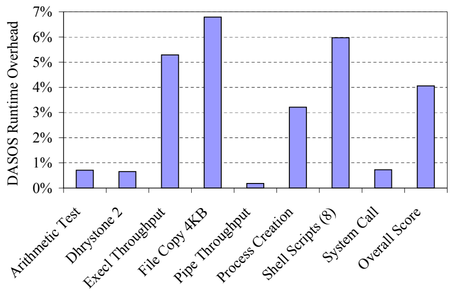


Fig. 6. Runtime Overhead of the Artificial Malware Immunization Infrastructure

Real-time Blocking of Malware Actions In addition to real-time detecting and reporting the first and all subsequent nonself system calls and nonsef commands of malware exploits, we can set the malware immunization infrastructure to block any nonself system call to defeat otherwise working exploits in real-time. Unlike randomization based approaches (e.g., ASLR) which usually cause the program under attack to crash with segmentation fault, the block mode of our artificial malware immunization framework can gracefully terminate the process/thread that has been attacked and compromised.

Real-time Forensics of Live Malware Actions Another unique feature of our artificial malware immunization framework is that it enables real-time forensics of live malware actions. We have developed a user space utility `dasosTSViewRT.sh` to identify and report the first and all the subsequent nonself system calls invoked, nonself commands issued by the attacker and nonseld processes spawned in real-time.

Our artificial malware immunization infrastructure can not only collect all the nonself system calls but also all the nonself processes and corresponding commands triggered by the exploit. We have developed a user level utility `dasosNewProcCmdRT.sh` to collect and report all those nonself processes and commands triggered by the exploit. Figure 5 shows all the nonself processes and corresponding commands of the working exploit of Snort 2.6.1. Specifically, the exploit first compromised the Snort process 5114 into a bourn shell, then issued commands "ps", "id", "ls", "ls" and "ifconfig" respectively. As with all dasos messages, the reported nonself processes and corresponding commands are timestamped at the precision of millisecond.

3.2 Performance Measurements

Our artificial malware immunization infrastructure intercepts and checks all system calls, and our transparent malware immunization tool instruments each function invocations in programs. To understand the performance impact of our artificial malware immunization, we have measured both the run-time overhead and the size overhead of our malware immunization infrastructure and our transparent malware immunization tool.

Our physical test platform is a MacBook Pro laptop with one 2.4GHz Intel Core 2 Duo CPU and 2GB memory running Mac OS X 10.5.8. The Fedora Core 6 Linux is running inside a vmware fusion virtual machine with 512MB RAM. For all our run-time performance measurements, we have used the average result of 10 independent runs.

We have used UnixBench to measure the run-time performance overhead of our artificial malware infrastructure over the original 2.6.18.2 Linux kernel. Specifically we ran UnixBench 10 times in our artificial malware immunization infrastructure and the original 2.6.18.2 Linux kernel and used the average to calculate the run-time performance overhead. Figure 6 shows the run-time overhead of 9 benchmarks of UnixBench. The highest run-time performance overhead is 6.8% for file copy with 4KB buffer size. The second highest overhead is 6% for shell scripts. On the other hand, the overheads for process creation and system call are 3.2% and 0.7% respectively. The overall UnixBench overhead is 4%. Therefore, our artificial malware immunization incurs neglectable run-time performance overhead.

4 Related Works

Models of self Our work is closely related to how to model the "self" of programs. Forrest et al. [15, 14] have pioneered in applying the idea of immunology to computer security. In their seminal paper [15], they have first demonstrated that the "self" for Unix processes can be approximately expressed by short sequences of system calls obtained from run-time traces of the protected process, and such a sense of self can be used to build intrusion anomaly detection systems. A number of followup works [35, 29, 13] have improved the effectiveness of the system call sequence based models of self in the context of intrusion anomaly detection. Model-carrying code [30] builds security-relavant model of self upon not only the system call sequence (in form of FSA) but also the value and relationship of system call arguments. Wagner and Dean [32] have first demonstrated that it is indeed possible to build a "complete" (but not necessarily pure) system call sequence based model of self via static analysis of the program source code such that there will be no false positive (there will be false negative though) in detecting the nonself. A number of later works [17] have improved the effectiveness and run-time efficiency of the models of self built from static analysis.

Since the program control flow graph contains all legitimate system call sequences, the control flow graph is a more precise expression of the "self" of the program. A number of methods [22, 5, 4, 18, 26] have been proposed to enforce the run-time control flow integrity of the protected program at different precisions. However, it is more difficult to model and enforce the complete control flow than system call sequence at run-time, especially when there are dynamically linked or share libraries involved. To mitigate data flow attacks [11], researchers have proposed modelling system call arguments [8] or the data flow graph [10]. In a sense, these models of run-time data flow can be thought as part of the normal behavior or "self" of the program.

Our model of dynamically assigned sense of self differs from all these works in that it actively assigns a unique mark to the protected program rather than passively reflecting the inherent nature (e.g., system call sequence, control or data flow) of the program. While authenticated system call [27, 19] actively assigns the authentication code to each system call, it does not form any sense of self of the program in that it can not distinguish different programs if they call any shared library functions.

Malware defense Stack guard [12], stack ghost [16] and windows vaccination [25] prevent stack based overflow by protecting the return address from being modified by the malware. However, they are not effective against other attack vectors such as heap based overflow. Linn et al. [24] proposed a method to prevent injected code from invoking system calls by checking the system call instruction address at run-time against previously recorded locations of legitimate system call instructions. While such a method can effectively block the system calls triggered by the injected code, it is not effective against return-to-libc attack where the malware calls existing libc functions rather than the system call.

Randomization based approaches [20, 21, 6, 7, 28, 34] protect applications and systems via randomizing the instruction set, address space layout or address-like strings in the packet payload. However, they can not detect the malware before it crashes the vulnerable application. In addition, it is difficult to apply the instruction set randomization (ISR) based approaches to shared libraries which are supposed to be shared by multiple applications. In contrast, our proposed malware immunization based on dynamically assigned sense of self can detect malware before it crashes the vulnerable applications. This unique capability enables new forensics analysis functionalities that were not possible before. Furthermore, our malware immunization framework is amenable to dynamically linked and shared libraries.

5 Conclusions

Inspired by the self-nonself discrimination in natural immune systems, we have explored a new direction in building the artificial malware immunization. Our malware immunization framework is based on dynamically assigned sense of self, which is essentially a randomly generated unique mark assigned to each system call of the immunized program. Such a dynamically assigned sense of self enables us to effectively and efficiently distinguish the unmarked nonself (i.e., malware action) from the marked self.

Specifically, our artificial malware immunization framework is able to detect the first and all the subsequent nonself system calls and nonself commands issued by almost all application level malwares. This unique capability enables not only the real-time detection, blocking of malware, but also the real-time forensics of the live malware actions. Since our malware immunization does not require any specific knowledge of the malware, it could be effective against previously unknown malwares.

We have implemented the prototype of our artificial malware immunization framework and tested it with both code injection and the return-oriented exploit [31]. Our transparent malware immunization tool has successfully instrumented the GNU standard C library (glibc 2.5 of about 1 million lines of C code) and immunized real-world, vulnerable applications (e.g., Snort 2.6.1 of 140,000 lines of C code) against otherwise working exploits. Our experiments also show that our malware immunization incurs about 4% run-time performance overhead. These empirical results demonstrate the promise of our artificial malware immunization framework in protecting real-world, vulnerable applications from malwares.

References

1. ghttpd Daemon Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/2879.
2. Snort Open Source Network Intrusion Prevention and Detection System (IDS/IPS). http://www.snort.org/.
3. Snort/Sourcefire DCE/RPC Packet Reassembly Stack Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/22616.
4. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, pages 340–353. ACM, November 2005.
5. T.-c. C. Alexey Smirnov. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS 2005)*, February 2005.
6. E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 281–289. ACM, October 2003.
7. G. Barrantes, D. Ackley, S. Forrest, and D. Stefanovic. Randomized Instruction Set Emulation. *ACM Transactions on Information Systems Security (TISSEC)*, 8(1):3–40, 2005.
8. S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow Anomaly Detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE, May 2006.
9. F. M. Burnet. *Self and Not-Self*. Cambridge University Press, Cambridge, 1969.
10. M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 147–160, November 2006.
11. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, August 2005.
12. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78. USENIX, Auguest 1998.
13. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P 2003)*. IEEE, May 2003.
14. S. Forrest, S. Hofmeyr, and A. Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, 1997.
15. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P 1996)*. IEEE, May 1996.
16. M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66. USENIX, August 2001.
17. J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-Sensitive Intrusion Detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, September 2005.
18. R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient Intrusion Detection using Automaton Inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005)*. IEEE, May 2005.
19. T. Jim, R. D. Schlichting, M. Rajagopalan, and M. A. Hiltunen. System Call Monitoring Using Authenticated System Calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3):216–229, July 2006.

20. Z. K. Jun Xu and R. K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS 2003)*. IEEE, October 2003.
21. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 272–280. ACM, October 2003.
22. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206. USENIX, August 2002.
23. J. Klein. *Immunology: The Science of Self-Nonself Discrimination*. John Wiley & Sons, New York, 1982.
24. C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against Unexpected System Calls. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, August 2005.
25. D. Nebenzahl, M. Sagiv, and A. Wool. Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(1):78–90, Jan-Mar 2006.
26. N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2007)*. ACM, October 2007.
27. M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated System Calls. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*. IEEE, June 2005.
28. R. S. Sandeep Bhatkar and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, August 2005.
29. R. Sekar, M. Bendre, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, May 2001.
30. R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 15–28, October 2003.
31. H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*. ACM, October 2007.
32. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, May 2001.
33. D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*. ACM, October 2002.
34. X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet Vaccine: Black-box Exploit Detection and Signature Generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*. ACM, October 2006.
35. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P 1999)*, pages 133–145. IEEE, May 1999.
36. J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS 2003)*, Feburary 2003.